Parallelization of a Dynamic Unstructured Application using Three Leading Paradigms

Leonid Oliker

NERSC

Lawrence Berkeley National Laboratory

www.nersc.gov/~oliker

Rupak Biswas

MRJ Technology Solutions

NASA Ames Research Center

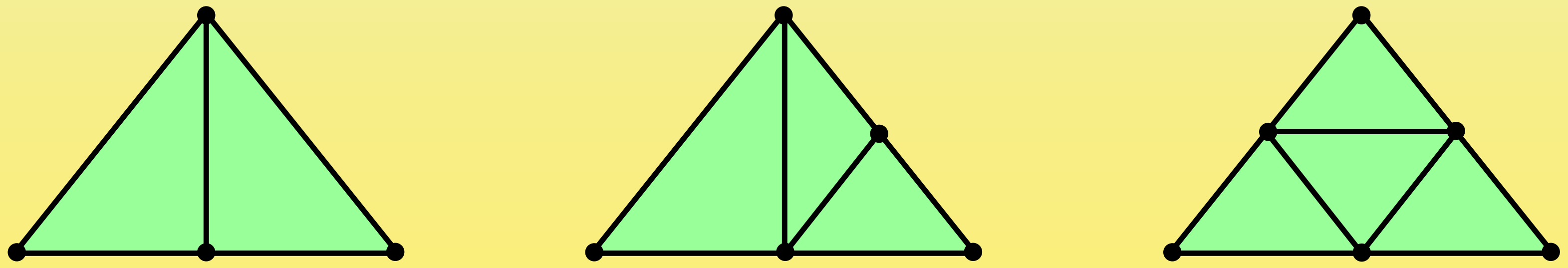
www.nas.nasa.gov/~rbiswas

Motivation and Objectives

- Real-life computational simulations generally require irregular data structures and dynamic algorithms
- Large-scale parallelism is needed to solve these problems within a reasonable time frame
- Several parallel architectures with distinct programming methodologies have emerged
- Report experience with the parallelization of a dynamic unstructured mesh adaptation code using three popular programming paradigms on three state-of-the-art supercomputers

2D Unstructured Mesh Adaptation

- Powerful tool for efficiently solving computational problems with evolving physical features (shocks, vortices, shear layers, crack propagation)
- Complicated logic and data structures
- Difficult to parallelize efficiently
  - Irregular data access patterns (pointer chasing)
  - Workload grows/shrinks at runtime (dynamic load balancing)
- Three types of element subdivision



Parallel Code Development

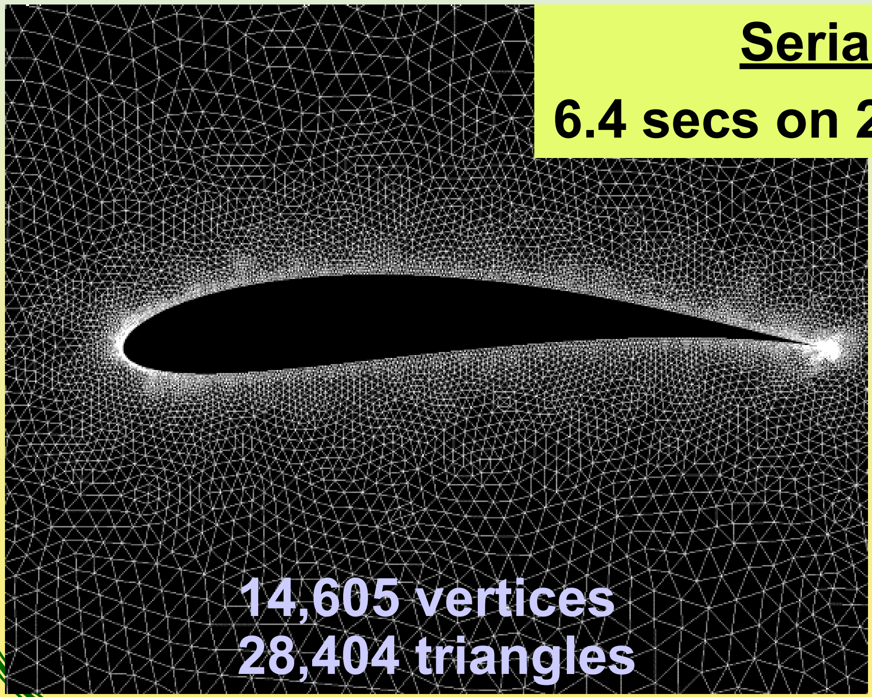
- Programming paradigms
  - Message passing (MPI)
  - Shared memory (OpenMP-style pragma compiler directives)
  - Multithreading (Tera compiler directives)
- Architectures
  - Cray T3E
  - SGI Origin2000
  - Tera MTA
- Critical factors
  - Runtime
  - Scalability
  - Programmability
  - Portability
  - Memory overhead

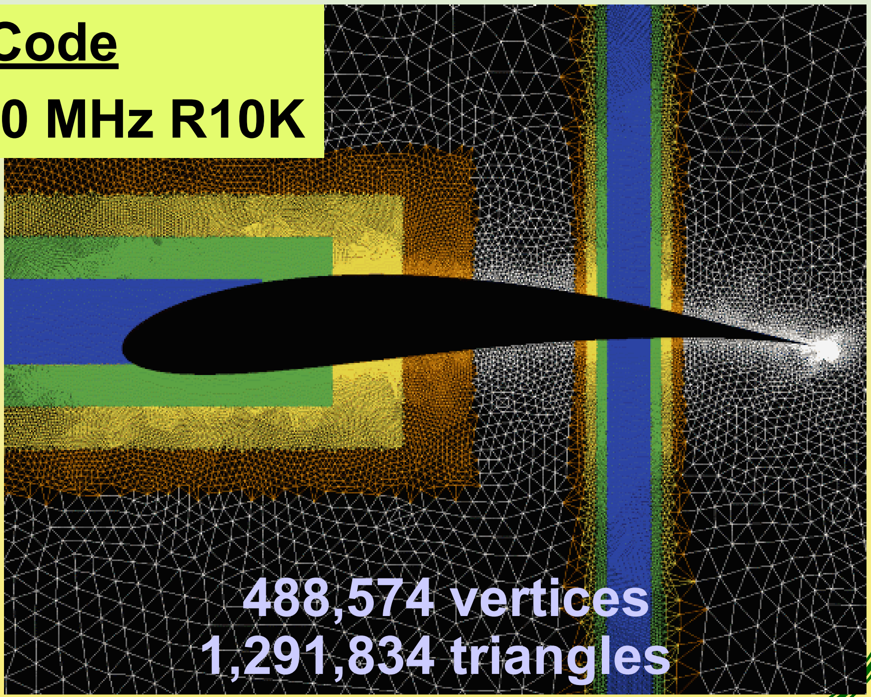
Test Problem

- Computational mesh to simulate flow over airfoil
- Mesh geometrically refined 5 levels in specific regions to better capture fine-scale phenomena

Serial Code

6.4 secs on 250 MHz R10K

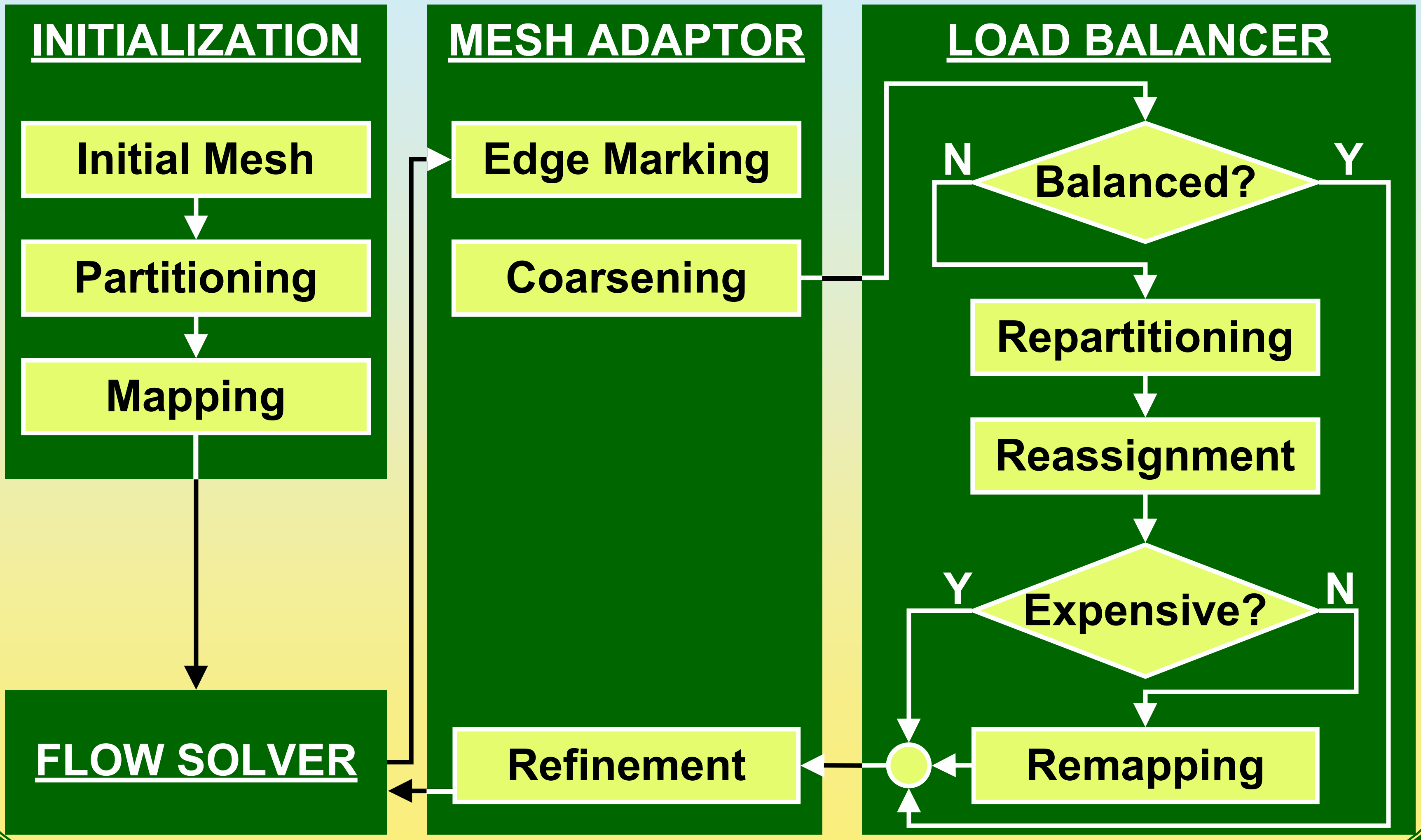




Distributed-Memory Implementation

- 512-node T3E (450 MHz DEC Alpha procs)
- 32-node Origin2000 (250 MHz dual MIPS R10K procs)
- Code implemented in MPI within PLUM framework
  - Initial dual graph used for load balancing adapted meshes
  - Parallel repartitioning of adapted meshes (ParMeTiS)
  - Remapping algorithm assigns new partitions to processors
  - Efficient data movement scheme (predictive & asynchronous)
- Three major steps (refinement, repartitioning, remapping)
- Overhead
  - Programming (to maintain consistent D/S for shared objects)
  - Memory (mostly for bulk communication buffers)

Overview of PLUM



Performance of MPI Code

- More than 32 procs required to outperform serial case
- Reasonable scalability for refinement & remapping
- Scalable repartitioner would improve performance
- Data volume different due to different word sizes

		Time (secs)				Data Vol (MB)	
System	P	Refine	Partition	Remap	Total	Max	Total
T3E	8	4.53	1.47	12.97	18.97	68.04	286.80
	64	0.78	1.49	1.81	4.08	6.88	280.30
	160	0.61	1.70	0.69	3.00	4.24	284.41
	512	0.14	4.70	0.25	5.09	0.99	310.40
O2K	2	13.12	1.30	24.89	39.31	50.11	60.64
	8	8.31	1.39	10.23	19.93	30.21	151.75
	64	1.41	2.30	1.69	5.40	4.17	132.34

Shared-Memory Implementation

- 32-node Origin2000 (250 MHz dual MIPS R10K procs)
- Complexities of partitioning & remapping absent
  - Parallel dynamic loop scheduling for load balance
- GRAPH_COLOR strategy (significant overhead)
  - Use SGI's native pragma directives to create IRIX threads
  - Color triangles (new ones on the fly) to form independent sets
  - All threads process each set to completion, then synchronize
- NO_COLOR strategy (too fine grained)
  - Use low-level locks instead of graph coloring
  - When thread processes triangle, lock its edges & vertices
  - Processors idle while waiting for blocked objects

Performance of Shared-Memory Code

- Poor performance due to flat memory assumption
- System overloaded by false sharing
- Page migration unable to remedy problem
- Need to consider data locality and cache effects to improve performance (require partitioning & reordering)
- For GRAPH_COLOR
  - <u>Cache misses</u> 15 M (serial) to 85 M (P=1)
  - <u>TLB misses</u> 7.3 M (serial) to 53 M (P=1)

	GRAPH_COLOR			NO_COLOR
P	Refine	Color	Total	Total
1	20.8	21.1	41.9	8.2
4	17.5	24.0	41.5	21.1
8	17.0	22.6	39.6	38.4
16	17.8	22.0	39.8	56.8
32	23.5	25.8	49.3	107.0
64	42.9	29.6	72.5	160.9

Multithreaded Implementation

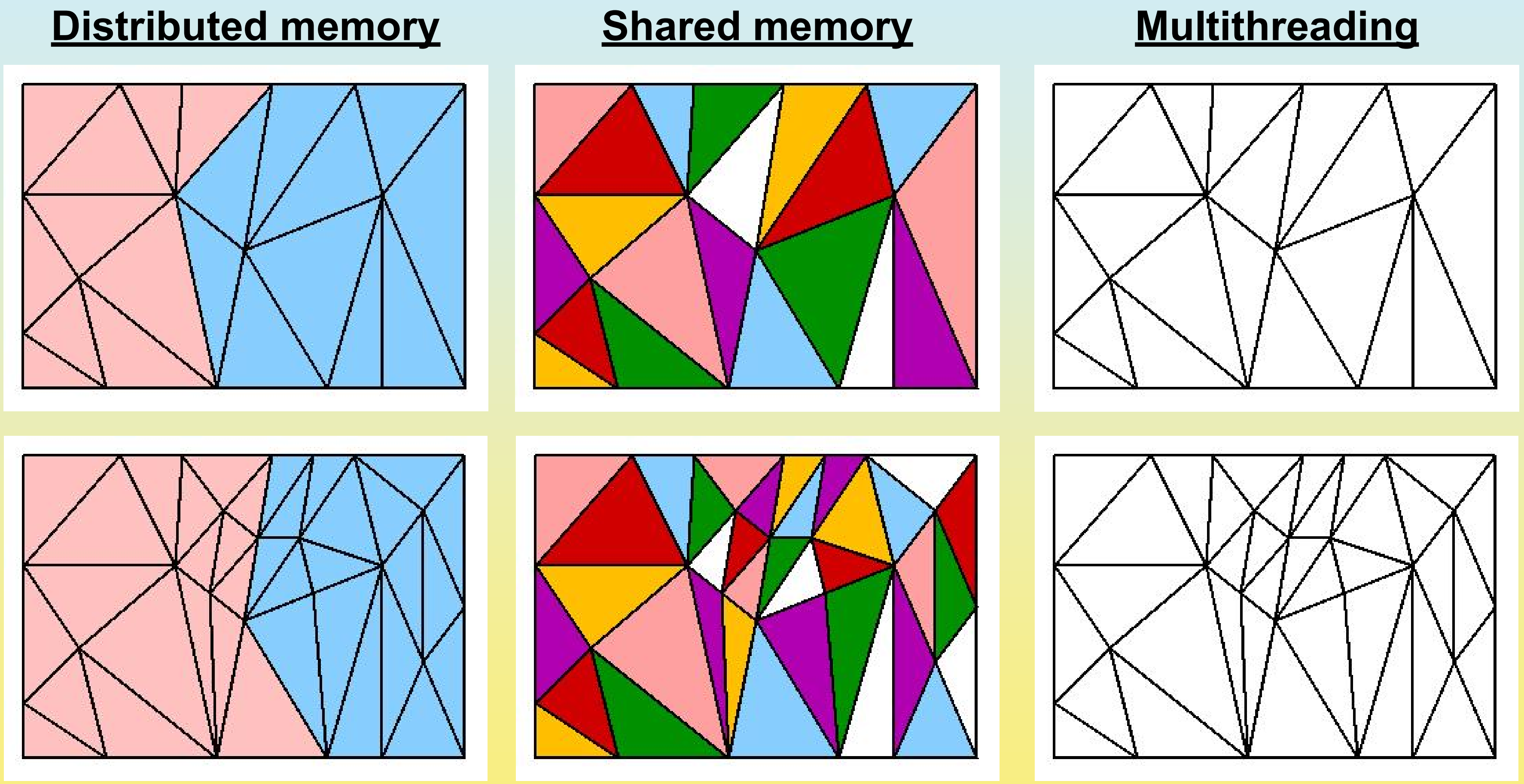
- 8-processor 250 MHz Tera MTA
  - 128 streams/proc, flat hashed memory, full-empty bit for sync
  - Executes pipelined instruction from different stream at each clock tick
- Dynamically assigns triangles to threads
  - Implicit load balancing
  - Low-level synchronization variables ensure adjacent triangles do not update shared edges or vertices simultaneously
- No partitioning, remapping, graph coloring required
  - Basically, the NO_COLOR strategy
- Minimal programming to create multithreaded version

Performance of Multithreading Code

- Sufficient instruction level parallelism exists to tolerate memory access overhead and lightweight synchronization
- Number of streams changed via compiler directive

	Streams per processor				
P	1	40	60	80	100
1	150.1	3.82	2.72	2.22	2.04
2		1.98	1.40	1.15	1.06
4		1.01	0.74	0.64	0.59
6		0.69	0.51	0.43	0.40
8		0.55	0.41	0.37	0.35

Schematic of Different Paradigms



Before and after adaptation (P=2 for distributed memory)

Comparison and Conclusions

Program Paradigm	System	Best Time	P	Code Incr	Mem Incr	Scala-bility	Porta-bility
Serial	R10000	6.4	1				
MPI	T3E	3.0	160	100%	70%	Medium	High
MPI	O2K	5.4	64	100%	70%	Medium	High
Shared-mem	O2K	39.6	8	10%	5%	None	Medium
Multithreading	MTA	0.35	8	2%	7%	High*	Low

- **Different programming paradigms require varying numbers of operations and overheads**
- **Multithreaded systems offer tremendous potential for solving some of the most challenging real-life problems on parallel computers**